Rewriting System for Profile-Guided Data Layout Transformations on Binaries

Olivier Aumage, Christopher Haine, Denis Barthou

Rewriting System for Profile-Guided Data Layout Transformations on Binaries

Christopher Haine, Olivier Aumage, and Denis Barthou

Univ. Bordeaux, LaBRI / Inria, France
Bordeaux INP, France

Abstract. Careful data layout design is crucial for achieving high performance. However exploring data layouts is time-consuming and error-prone, and assessing the impact of a layout transformation on performance is difficult without performing it. We propose to guide application programmers through data layout restructuring by providing a comprehensive multidimensional description of the initial layout, built from trace analysis, and then by giving a performance evaluation of the transformations tested and an expression of each transformed layout. The programmer can limit the exploration to layouts matching some patterns. We apply this method to two multithreaded applications. The performance prediction of multiple transformations matches within 5% the performance of hand-transformed layout code.

Keywords: Performance Tuning, Data Layout Restructuring, Memory Traces

1 Introduction

Adapting data allocations and structures to the way data is used is a key optimization for parallel architectures. Changing data layout can enhance spatial data locality and memory consumption, having a large impact on code performance. Associated with instruction rescheduling and loop nest transformations, layout restructuring has a strong impact on vectorization and may lead to a better use of cache hierarchy, through temporal and spatial locality. Data restructuring is a global optimization in general, requiring interprocedural analysis, and in languages such as C, possible aliases hamper the scope of transformations. When considering combined data layout and control-flow transformation, dependence analysis further limits the applicability of the methods. Finally, due to the complexity of memory hierarchy, the impact on performance of a data structure change is difficult to assess. To illustrate this difficulty, the simple choice between an array of structures (AoS) or a structure of arrays (SoA) is highly dependent on the use of the structure. Depending on the locality of data, it may be beneficial to use the SoA version if when using a single field at a time or the AoS version when using multiple fields (such as a complex number, for instance). For a parallel code, an Array of Structures of Vectors/Arrays may have to be considered, resulting in portability issues and unacceptable program complexity for the human programmer [16].

Several works have studied data layout restructuring for specific applications [21, 12] and for stencils [9]. In a recent work [1] the authors proposed a framework to analyze binary codes, and to formulate user-targeted hints about SIMDization potentials and hindrances. These hints provide the user with possible strategies to remove SIMDization hurdles, such as code transformations or data restructuring. However, this preliminary work conducted a qualitative analysis only, thus lacking an estimation in the transformation gains. In [7], we proposed a more quantified approach, to detect simple arrays and structures from execution traces, and suggest promising data layout transformations.

This paper proposes a novel approach for data restructuring. A formalization of data structures and of their transformations is described, independently of any control-flow or rescheduling optimization. We show that this framework can be used from memory traces in order to provide a quick assessment of potential gains (or lack of) to be expected from some transformations. For this purpose, we show how to setup mock-up executions for an application, in order to evaluate the impact of the transformation without the need to actually change the whole data structures or re-execute the whole application. This approach is evaluated on two real applications parallelized with OpenMP, combining restructuring and vectorization. The contributions proposed in this paper are the following:

- Description of data structure layouts and their transformations, independently of control-flow optimizations;
- Generation of mock-up codes with restructured layouts;
- Performance evaluation of mock-ups, with and without SIMDization.

The paper is organized as follows: Section 2 presents two motivating examples with sub-optimal data layout. Section 3 describes a method for finding an initial multidimensional layout matching a trace, and the possible transformations. Section 4 presents the evaluation methodology. The experimental results are discussed in Section 5 and Section 6 presents related work.

2 Motivating Examples

From a user perspective, abstract data types correspond to algorithmic requirements, but choosing the actual data layout requires to take compiler, runtime support and architectural constraints into consideration. We illustrate this gap between the data layout chosen for the two following applications. In the cardiac wave simulation [22], the hot spot of the OpenMP version of the application uses a large 4D array to store the whole data structure, as shown in Figure 2. The first 2 dimensions have starting index of 1, creating unnecessary gaps between lines. The third dimension is used as a structure with numbered fields, and the fourth dimension has a spatial locality issue, since it is indexed with the parity of the computation step (to keep only the previous computation results). While reordering dimensions here is not very complex, the ordering and locality choices for the last dimension depend on the computation itself and on the architecture.

```
for (X=1; X<Nx+1; ++X){
 for (Y=1; Y<Ny+1; ++Y){
   datarr[X][Y][13][(step-1)%2] =
    datarr[X][Y][13][(step-1)%2]
    -(2*Dp[0][0]+2*Dp[1][1])
    *datarr[X][Y][0][step%2]
    -Dp[1][0]*datarr[X-1][Y+1][0][step%2]
    +Dp[1][1]*datarr[X][Y+1][0][step%2]
    +Dp[1][0]*datarr[X+1][Y+1][0][step%2]
    +Dp[0][0]*datarr[X+1][Y][0][step%2]
    -Dp[1][0]*datarr[X+1][Y-1][0][step%2]
    +Dp[1][1]*datarr[X][Y-1][0][step%2]
    +Dp[1][0]*datarr[X-1][Y-1][0][step%2]
    +Dp[0][0]*datarr[X-1][Y][0][step%2];
}}
```

(a) Cardiac wave simulation excerpt

```
for(iL=0 ; iL < L/2 ; iL+=1) {
  for (j=0;j<4;j++) {
    r0 =  U[idn[4*iL]][0]*tmp[j];
    r0 += U[idn[4*iL]][1]*tmp[n2+j];
    r0 += U[idn[4*iL]][2]*tmp[2*n2+j];
    r1 =  U[idn[4*iL]][3]*tmp[j];
    r1 += U[idn[4*iL]][4]*tmp[n2+j];
    r1 += U[idn[4*iL]][5]*tmp[2*n2+j];
    r2 =  U[idn[4*iL]][6]*tmp[j];
    r2 += U[idn[4*iL]][7]*tmp[n2+j];
    r2 += U[idn[4*iL]][8]*tmp[2*n2+j];
    ID2[j] += r0 ;
    ID2[n2+j] += r1;
    ID2[2*n2+j] += r2;
  }
```

(b) Lattice QCD simulation excerpt

Fig. 1. Two examples of codes needing data layout restructuring. In the Cardiac wave simulation, the 4-D array `datarr` is used as an array of structures. For the QCD simulation, all elements are `complex double` values. The space iterated by the outer loop is a 4-D linearized space and the indirection used for U walks through the white elements of a 4-D checkerboard.

The second example considered is a Lattice QCD application, based on ETMC simulation[2]. The hotspot of the application performs several matrix-vector computations. Each matrix is described as an element of a large array, `U`. The space iterated by `iL` is a 4D linearized space. In this 4D space, only the white elements of a checkerboard are accessed, through an indirection array. Deciding how to restructure this array and whether it is worth to get rid of the indirection is important for the code performance. This example is difficult to analyze statically and would require some additional information from the user. An analysis based on traces on the contrary would capture the regularity of the accesses, in spite of the indirection.

3 Layout Description and Transformations

We give here a formal description for layouts and rules for transforming them.

```
for i0 = 0 to 255
  for i1 = 0 to 255
    val 0x00001000 + 16384*i0 + 32*i1
    val 0x00001008 + 16384*i0 + 32*i1
  endfor
  for i1 = 0 to 255
    val 0x00003010 + 16384*i0 + 32*i1
    val 0x00003018 + 16384*i0 + 32*i1
  endfor
endfor
```

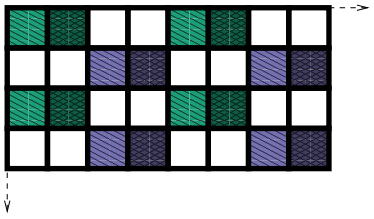

Fig. 2. Example trace for Qiral for array `U` accesses, simplified 2D version for conciseness. Each color in the map represents one line of the trace.

3.1 Data Layout Description

Data structures are considered as any combination of arrays and structures, of any length. A layout is the description of this structure and of the elements that are accessed in it. A layout can be defined only for a limited code fragment. When considering a syntactic memory access expression in the code, it defines a set of memory address values. This set can be denoted as $base + I$ where $base$ is the base address and I is the set of positive integers, including 0. All addresses are within a range $[base, base + d - 1]$ where d is the diameter of I. The set of offsets I can be represented by a layout function $S_{I,d}$, characterizing I:

$$\begin{aligned} S_{I,d} : [0, d-1] &\to \{0, 1\} \\ x &\to 1 \text{ if } x \in I, 0 \text{ otherwise} \end{aligned}$$

$S_{I,d}$ is called a *structure layout*. If $I = [0, d - 1]$ (all elements are accessed), $S_{[0,d-1],d}$ is more specifically called an *array layout*, denoted A_d. Note that these terms of arrays and structures may not correspond to the structures really occurring in the source code. To build a multidimensional data structure, we define the product operator $\otimes$ and the sum $\oplus$ on layout functions L_1 and L_2:

$$\begin{aligned} L_1 \otimes L_2 : I_1 \times I_2 &\to \{0, 1\} \\ x, y &\to L_1(x) * L_2(y) \end{aligned} \qquad \begin{aligned} L_1 \oplus L_2 : I &\to \{0, 1\} \\ x &\to L_1(x) + L_2(x) \end{aligned}$$

For the product, the two layout functions L_1 and L_2 may have two different domains, I_1 and I_2. For the sum, the domain of the two functions must be the same. The + operation is a saturated addition between integers. With this notation, Array of Structures correspond to the combination of the two types of layout, described by $A_{d'} \otimes S_{I,d}$ for some values of d, d' and I. The formal description corresponds to the intuitive representation of the data. The same factorization identities exist with $\oplus$ and $\otimes$ as with integers. Some simplifications are possible between expressions involving both operators:

$$(S_{I,d} \otimes L) \oplus (S_{J,d} \otimes L) = S_{I \cup J,d} \otimes L, \quad (L \otimes S_{I,d}) \oplus (L \otimes S_{J,d}) = L \otimes S_{I \cup J,d}$$

3.2 Finding the Initial Multidimensional Layout

In the general case, the memory accesses are given as flat, linearized addresses. The objective of this section is to find out the different multidimensional layouts used in the code fragment considered. On the source code, finding whether two memory accesses correspond to the same array region correspond to an alias analysis. Delinearization can be used in some simple cases to retrieve the multidimensional structure associated to the addresses. Because indirections or complex operations can be involved in the address computation, as shown in the two codes given as motivating examples, we propose in this paper to resort to memory traces. The code fragment is executed and all memory accesses generate a trace. This trace is compacted on-the-fly with the NLR method [11] in order to find possible recurring stride patterns. The following rewriting system

transforms a flat layout into a multidimensional layout:

$$S_{I,m} \rightarrow \ S_{J,n} \otimes A_p \text{ if } I = \{j * p + k, j \in J, k \in [0, n]\} \quad (1)$$

$$S_{I,m} \rightarrow \ A_n \otimes S_{J,p} \text{ if } I = \{k * p + j, j \in J, k \in [0, n]\} \quad (2)$$

$$S_{n*I+p,m*n} \rightarrow \ S_{I,m} \otimes S_{\{p\},n} \text{ if } p < n \quad (3)$$

with $n * I + p = \{n * i + p, i \in I\}$. The first rule corresponds to the case where the initial layout is a structure of array, the second to an array of structure, and the third is the general case, where two structures layouts have been linearized. The initial multidimensional layout can be found by applying these rules iteratively until convergence. The rewriting system is confluent and convergent. Convergence comes from structures with diminishing sizes. We assume that array layouts are not rewritten. Confluence entails that the rules can be applied in any order and results from the fact that there is only one way to rewrite any given part of the addresses.

We apply the previous algorithm to restructure the trace given in Figure 2. The trace is given as a `for..loop` enumerating addresses and out of simplification, is a simplified version for the memory access of matrix U (2D case, only first statement, no outer dimension). The following initial structure corresponds to the set of values accessed by the trace:

$$\begin{array}{l} U + S_{\{2048*[0,255]+4*[0,255]\},d} \\ \quad \oplus S_{\{2048*[0,255]+4*[0,255]+1\},d} \\ \quad \oplus S_{\{2048*[0,255]+4*[0,255]+1024+2\},d} \\ \quad \oplus S_{\{2048*[0,255]+4*[0,255]+1024+3\},d} \end{array} \Rightarrow \begin{array}{l} U + A_{256} \otimes S_{\{0\},2} \otimes A_{256} \otimes S_{\{0,1\},4} \\ \quad \oplus A_{256} \otimes S_{\{1\},2} \otimes A_{256} \otimes S_{\{2,3\},4} \end{array}$$

Applying Rule 3, then merging the first two lines and the last two, and then applying Rule 2 and finally Rule 1 leads to the formulation on the right. This corresponds to an AoSoAoS: This is an array of 2 lines, even lines and odd lines. Even lines have 256 elements that are structures of 4 doubles, using only the first 2. Odd lines have 256 elements having 4 doubles, using only the last 2. This is represented in Figure 2.

3.3 Transformations

We define layout transformations as rewriting rules that rewrite the layouts defined in previously defined formalism. For these rules, rules applying to structures S are assumed not to apply to arrays. If $S_{I,d}(x) = 1$, we will rather write it as: S_d. $\#I$ corresponds to the number of elements in I:

$$L \otimes L' \rightarrow L' \otimes L \quad (4) \qquad S_{I,n} \otimes S_{J,m} \rightarrow S_{I',n*m},\ if \#I' = \#I \times \#J \quad (6)$$

$$A_{n*m} \rightarrow A_n \otimes A_m \quad (5) \qquad S_{I,d} \rightarrow S_{I',d'},\ if \#I = \#I', d \leq d' \quad (7)$$

Rule 4 permutes two layouts, Rule 5 cuts an array in two arrays. Rule 6 fusions two structure layouts and the last rule, 7, removes unused elements in a structure. In a layout expression composed of different terms in a $\oplus$, all terms of the

sum at the same position must be rewritten with the same rule, since it corresponds to the same sub-structure. All transformations preserve the number of elements in the layouts.

3.4 Exploring Layouts

The previous rewriting system generates a finite but potentially large number of layouts. We propose a strategy to limit the exploration. Rule 5 is only applied at most once to split an array for SIMDization purposes. One of the created array is then permuted in the rightmost position of the term in order to create a possible vector of elements. Rule 7 is applied whenever possible. Rule 6 simplifies code generation by fusing contiguous dimensions. This rule is only applied at the end of the rewriting.

To further reduce exploration, we propose to guide the generation by proposing patterns of layouts. For instance, SIMDization requires that the layout ends with an array. Only terms in the form of the regular expression $* \otimes A$ are considered. For instance, on the two examples shown as motivating examples, we look for layouts of the form $* \otimes A$ or $* \otimes A \otimes S_c$ (with S_c the structure corresponding to complex numbers). This leads to the layouts presented in the following table:

Code	Initial Layout	Transformed Layouts [short name]
Qiral excerpt	$A_L \otimes S_{\{1\},d} \otimes S_m \otimes S_c$	$A_{L/v} \otimes S_{m\times c} \otimes A_v$ [AoSoA-dbl]
Qiral precond. excerpt	$\bigoplus_{x,y,z,t\equiv 0[2]} \left(\bigotimes_{k=x,y,z,t} A_l \otimes S_{k,2} \right) \otimes S_{\{1\},d} \otimes S_m \otimes S_c$	$A_{L/v} \otimes S_m \otimes A_v \otimes S_c$ [AoSoA-cplx] $S_{m\times c} \otimes A_L$ [SoA-dbl] $S_m \otimes A_L \otimes S_c$ [SoA-cplx]
Qiral application	$A_L \otimes S_d \otimes S_m \otimes S_c$	$A_{L/v} \otimes S_d \otimes S_{m\times c} \otimes A_v$ [AoSoA] $S_d \otimes S_m \otimes A_L$ [SoA]
Cardiac Wave	$A_X \otimes A_X \otimes S_{\{0\},a} \otimes S_{\{0\},s}$	$A_X \otimes A_X$

The preconditioned version for QIRAL has a detected 4D checkerboard pattern, here expressed in a concise form, and v has the size of a SIMD vector. Checkerboard compression leads to the same transformations, only with L half the size. The QIRAL excerpt corresponds to the code presented as the motivating example while the application includes a larger scope of code.

4 Transformation Evaluation

This section deals with the quantified part of the user feedback we provide. The idea is to estimate the potential speedup of transformations in order to help the user make a choice for data restructuring.

4.1 Principle of Mock-up Evaluation

We propose an evaluation methodology that explores a set of different layout transformations. Because these transformations are based on the values collected by memory traces, the generated transformed codes are in general not semantically equivalent to the initial code, outside of the trace. However they

can serve as performance mock-ups. The idea is to measure possible performance gains of the application by executing the mock-ups. To preserve the application execution conditions, the mock-up is executed in the context of the application. Checkpoint/restart technique is used for this objective: Assuming the user knows the hotspot of the application, the original binary code is patched with a checkpoint right before the hotspot and then run until the checkpoint is reached. This checkpoint generates an execution context, used for capturing the trace and running/evaluating the mock-ups. The binary code is instrumented in order to collect the memory trace and restarted from this context. Then several layout transformations are applied on the initial code, generating new versions of the code that are restarted from the same context. As the checkpoint/restart mechanism preserves the memory addresses in use, the addresses and sizes of layouts captured in the trace can be reused in the mock-up codes. We rely on this property for generating data layout copies and the transformed codes. Our approach does not preserve however the hotspot cache state. Cache warm-up may be a solution to this issue, but goes beyond the scope of this paper. Mock-ups are stopped when the control leaves the hotspot and the timing is deduced at this point. For checkpoint/restart, we resort to the BLCR library [8].

4.2 Automatic Mock-up Generation Technique

Mock-ups are generated at compile time, as library functions. A mock-up corresponds to the initial hotspot, with different memory accesses and their address computation. The rest of the computation itself corresponds to the original code.

Before executing the mock-up, the data layout has to be created and data copied. This copy-in operation is guided by the trace information. The objective is to optimize the hotspot performance, and to push away the copies from the kernel to minimize their impact, avoiding cache pollution due to the copy itself. We choose to move the copy up to the beginning of the function if applicable, the limit being the last write on the array we want to restructure. This is determined automatically by trace inspection.

The sequence of transformation rules applied to the initial layout corresponds also to transformations on the iterators of these structures. The copy codes are simple loops changing one layout, with one iterator, into another. For the indexation of data in the computation code, the control is kept unchanged. New scalar iterators are created in order to map the previous index to the new index. For this, the trace provides for each individual assembly instruction the sequence of addresses accessed. This sequence of indices is transformed into a sequence of new indices, of the new layout.

The binary code is parsed with the MAQAO tool, and the modified code of the mock-up is generated in a C file, using assembly inline. The advantage of this approach is to rely on the compiler for an optimized register allocation for all the new induction variables added for indexing, and for removing dead code. For instance, the loads corresponding to the indirection are removed when reindexing the data structure in a simpler way. The code generated is only valid within the scope of the values collected by the trace.

4.3 Combining Layout Restructuring with SIMDization

Data restructuring is a SIMDization-enabling transformation, as data can be placed contiguously to fill a vector. We perform SIMDization whenever dependences allows it, impacting the control (loops) of the hotspot. From the trace analysis, we build a dependence graph that determines whether some arrays can be vectorized. We rely on MAQAO for this analysis [1], as well as for the detection of loop structures and for loop counters. The generated vectorized loop has a shorter loop trip count by a factor equal to the architecture vector size. This loop trip count is retrieved from the memory traces. All instructions involving the initial data structure have to be replaced by their vectorized counterpart, including load and stores. Some compiler optimization can be untangled, such as partial loads that are replaced by a single packed load operation. Reductions are detected through dependence graph analysis, and are replaced using horizontal operations. We detect read-only arrays or constants and unpack them. However, our SIMDization step from binary code to assembly code (assembly inline) is still fragile and essentially only applies a straightforward vectorization scheme.

5 Experimental Results

The objective of the section is to show how relevant the speedup hints are, in the sense that they provide useful advice to the programmer. To do so, we compare our mock-up speedups with the actual performance observed by restructuring by hand the C code, using layouts defined in Section 3.3. All experiments are conducted on an Intel(R) Xeon(R) CPU E5-2650 2GHz 2*8-core processor with SSE2 features, using icc 15.0.0 and gcc 5.3.1 compilers, both with `-O3` flag.

Lattice QCD: Figure 3 shows performance of both mock-ups and hand-transformed codes for the loop nest in Figure 1.(b). The hand-tuned code focuses only on restructuring layout and does not perform explicit SIMDization. It appears that gcc does not vectorize the code when handling `complex` data types and performs poorly even compared to the non-vectorized mock-ups. For the code without preconditioning (left graph), all mock-ups predict performance improvement for each of the four transformation presented, with an average relative error of 16% compared to the hand-tuned codes. For *AoSoA-cplx*, the mock-up underestimate performance. The reasons comes from the fact that `icc` optimizes the complex multiply and the load/stores, outperforming the naive SIMDization of the mock-up. Similar conclusions hold for the code with even/odd preconditioning. For the whole multithreaded hotspot function, the manually restructured version resorts to intrinsics, as the compilers do not manage auto-vectorization. Predictions for *SoA* and *AoSoA* are reliable with an average relative error of 4%, as shown in Figure 4, as mock-up SIMDization perform close to user restructured code. With a packed thread policy and hyper-threading disabled, the multithreaded context does not disrupt the mock-up prediction, since the code is parallel and compute-bound.

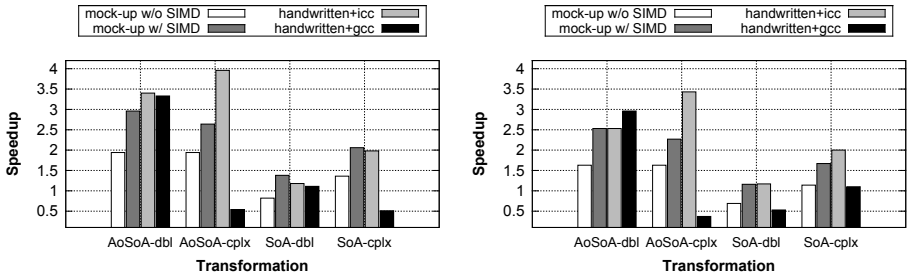

Fig. 3. Lattice QCD Benchmark without Preconditioning (left), with Even/Odd Preconditioning (right) Speedup, single thread.

2D Cardiac Wave Propagation Simulation: The hotspot is not vectorized, but it is successfully vectorized after data layout restructuring; Consequently no intrinsics are used in hand-tuned codes. We study layout restructuring impact on performance on two different datasets, corresponding to two different layout sizes. Speedups obtained after restructuring are shown in Figure 5 for the Dataset-256. With only the restructuring, the mock-ups exhibit a speed-up of 2.4× on average. When considering mock-ups with SIMDization, the gain of SIMDization alone is around 2×. Mock-up prediction average relative error is 9% too optimistic in this experiment. This over-estimation is explained by the effect of cache warm-up. In the mock-ups, the data copy loads data in the cache right before the hotspot. In the application, such "prefetch" is not performed. The input size is multiplied by a factor 4 using Dataset-512. In this new configuration, restructuring gain is dramatically higher than before, increasing with the number of threads and reaching roughly 14× with 8 threads, as application achieves to take full advantage of all private L2 caches. Moreover, prediction remains consistently slightly overoptimistic as memory cache may be warmer before kernel execution than actual real application cache, while still being accurate with an average relative error of as low as 5%.

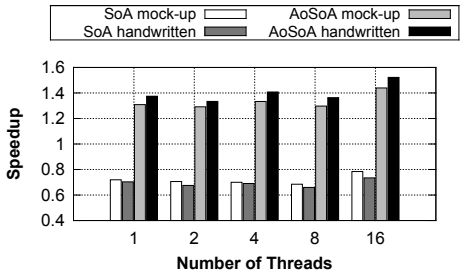

Fig. 4. Lattice QCD Application Restructuring+SIMD speedup with respect to thread number.

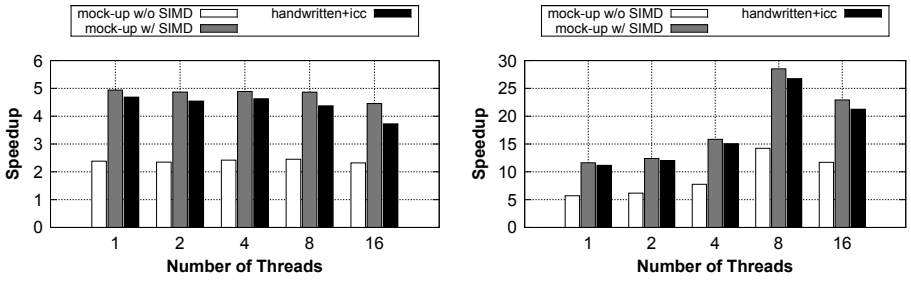

Fig. 5. 2D Wave Propagation Application Restructuring+SIMDization Speedup on Dataset-256 (left) or on Dataset-512 (right) — with respect to reference using respectively equal number of threads — average relative error is 9% ± 8% (left), 5% ± 2% (right)

6 Related Work

Many modern languages, in particular object oriented languages, propose a layer of abstraction between data types and the data layout in memory (hierarchical arrays, C++ libraries). However, few works propose to restructure existing data, in codes written in C or Fortran. This abstraction layer is also provided by libraries, hiding in particular the complexity of AoSoA layouts with SIMDization to the user (Cyme [5], Boost:SIMD or Kokkos [4], to name a few).

The StructSlim profiler [15] helps programmers into data restructuring through structure splitting. For GPU, copy is performed at transfer time and data layout change is also performed at this step [20, 18]. Code analysis is performed statically, on OpenCL for instance. The same approach has been explored for heterogeneous architectures [14], assessing affinity between fields and clustering fields, and devising multi-phase AoS vs SoA data layouts and transforms. VP^3[23] is a tool for pinpointing SIMDization related performance bottlenecks. It tries to predict performance gains by changing the memory access pattern or instructions. However, it does not propose high level restructuring. Similarly, ArrayTool [13] can be used to regroup arrays, to gain locality, but there is no deeper change in data layouts.

Annotations and specific data layout optimizations with compiler support has been proposed by Sharma *et al.* [17]. The source-to-source transformation requires to describe in a separate file the desired array interleaving. Similarly, the array unification described by Kandemir [10] and Inter-Array Data regrouping [3] propose to merge different arrays at compile-time in order to gain locality. The POLCA semantics-aware transformation toolchain is an Haskell framework offering numerous transformation operators using programmer inserted pragma annotations [19]. Neither of these approach provide an assessment of the performance gains to guide the user restructuring or hint generation, and these compile-time approaches cannot handle indirections.

Delinearization is the first analysis on the compiler side, in order to be able to restructure the layout. Parametric delinearization, for some particular codes, has been proposed by Grosser *et al.* [6]. Specifically for stencil codes, using the polyhedral model, Henretty *et al.*[9] propose a complete restructuring of layout for SIMDization. This would not apply to the Lattice QCD code with the even/odd preconditioning (indirection)

Compared to the authors previous work [7], the work presented in this paper gives a more general framework for the recognition of complex data layouts and systematic exploration of data layouts. The code generation and SIMDization are automatically achieved, for a given transformation.

7 Conclusion

We have presented in this paper an original contribution for assessing the impact on performance of data layout restructuring. The layout transformations, based on profile information and described by a rewriting system, can be shown and explained to the user, from the initial layout to the transformed one. These transformations can then be applied and explored directly on a binary code generating automatically a new binary code. A set of different restructuring has been combined with SIMDization and the evaluation has been conducted on two applications, with different parameters (size of input, preconditioning used) and using different number of threads. The results show that the performance prediction of mock-up restructuring is reliable compared to a hand-tuned transformation and SIMDization (below 5% in average of relative error).

References

1. O. Aumage, D. Barthou, C. Haine, and T. Meunier. Detecting simdization opportunities through static/dynamic dependence analysis. In *Workshop on Productivity and Performance (PROPER)*, 2013.
2. D. Barthou, G. Grosdidier, M. Kruse, O. Pene, and C. Tadonki. QIRAL: A High Level Language for Lattice QCD Code Generation. In *Programming Language Approaches to Concurrency and Communication-centric Software Workshop*, Tallinn, Estonia, 2012. arXiv:1208.4035.
3. C. Ding and K. Kennedy. Inter-array data regrouping. In *Intl. Workshop on Languages and Compilers for Parallel Computing*, pages 149–163, London, UK, UK, 2000. Springer-Verlag.
4. H. Edwards and C. Trott. Kokkos: Enabling performance portability across manycore architectures. In *Extreme Scaling Workshop*, pages 18–24, Aug 2013.
5. T. Ewart, F. Delalondre, and F. Schrmann. Cyme: A library maximizing simd computation on user-defined containers. In *Supercomputing*, volume 8488 of *LNCS*, pages 440–449. Springer, 2014.
6. T. Grosser, J. Ramanujam, L.-N. Pouchet, P. Sadayappan, and S. Pop. Optimistic delinearization of parametrically sized arrays. In *ACM on Intl. Conf. on Supercomputing*, pages 351–360, New York, NY, USA, 2015. ACM.

7. C. Haine, O. Aumage, E. Petit, and D. Barthou. Exploring and evaluating array layout restructuring for simdization. In *Intl. Workshop on Languages and Compilers for Parallel Computing*, pages 351–366, Cham, 2015. Springer.
8. P. H. Hargrove and J. C. Duell. Berkeley lab checkpoint/restart (blcr) for linux clusters. *Journal of Physics: Conf. Series*, 46(1):494, 2006.
9. T. Henretty, K. Stock, L.-N. Pouchet, F. Franchetti, J. Ramanujam, and P. Sadayappan. Data layout transformation for stencil computations on short-vector simd architectures. In *Intl. Conf. on Compiler Construction*, pages 225–245, Berlin, 2011. Springer-Verlag.
10. M. Kandemir. Array unification: A locality optimization technique. In *Compiler Construction*, volume 2027 of *LNCS*, pages 259–273. Springer, 2001.
11. A. Ketterlin and P. Clauss. Prediction and trace compression of data access addresses through nested loop recognition. In *ACM/IEEE Intl. Conf. on Code Generation and Optimization*, pages 94–103, New York, NY, USA, 2008. ACM.
12. M. Kong, R. Veras, K. Stock, F. Franchetti, L.-N. Pouchet, and P. Sadayappan. When polyhedral transformations meet SIMD code generation. In *ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation*, 2013.
13. X. Liu, K. Sharma, and J. Mellor-Crummey. Arraytool: A lightweight profiler to guide array regrouping. In *Intl. Conf. on Parallel Architectures and Compilation*, pages 405–416, New York, 2014. ACM.
14. D. Majeti, K. S. Meel, R. Barik, and V. Sarkar. Adha: Automatic data layout framework for heterogeneous architectures. In *Intl. Conf. on Parallel Architectures and Compilation*, pages 479–480, New York, 2014. ACM.
15. P. Roy and X. Liu. StructSlim: A lightweight profiler to guide structure splitting. In *ACM/IEEE Intl. Conf. on Code Generation and Optimization*, 2016.
16. N. Satish, C. Kim, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, and P. Dubey. Can traditional programming bridge the ninja performance gap for parallel computing applications? In *Intl. Symp. on Computer Arch.*, pages 440–451, Washington, DC, USA, 2012. IEEE.
17. K. Sharma, I. Karlin, J. Keasler, J. R. McGraw, and V. Sarkar. Data layout optimization for portable performance. In J. L. Träff, S. Hunold, and F. Versaci, editors, *Intl. Euro-Par Conference*, pages 250–262. Springer, 2015.
18. I.-J. Sung, G. Liu, and W.-M. Hwu. Dl: A data layout transformation system for heterogeneous computing. In *Innovative Parallel Comp., 2012*, pages 1–11, May 2012.
19. S. Tamarit, J. Mario, G. Vigueras, and M. Carro. Towards a semantics-aware transformation toolchain for heterogeneous systems. In *Program Transformation for Programmability in Heterogeneous Arch. Workshop*, 2016.
20. Y.-Y. Tseng, Y.-H. Huang, B.-C. C. Lai, and J.-L. Lin. Automatic data layout transformation for heterogeneous many-core systems. In C.-H. Hsu, X. Shi, and V. Salapura, editors, *Network and Parallel Computing*, volume 8707 of *LNCS*, pages 208–219. Springer, 2014.
21. B. Videau, V. Marangozova-Martin, L. Genovese, and T. Deutsch. Optimizing 3d convolutions for wavelet transforms on cpus with sse units and gpus. In *Intl. Euro-Par Conference*, 2013.
22. W. Wang, L. Xu, J. Cavazos, H. H. Huang, and M. Kay. Fast acceleration of 2d wave propagation simulations using modern computational accelerators. *PLoS ONE*, 9(1):1–10, 01 2014.
23. D. C. Wong, D. J. Kuck, D. Palomares, Z. Bendifallah, M. Tribalat, E. Oseret, and W. Jalby. Vp3: A vectorization potential performance prototype. In *Workshop on Programming Models for SIMD/Vector Processing*, Feb 2015.